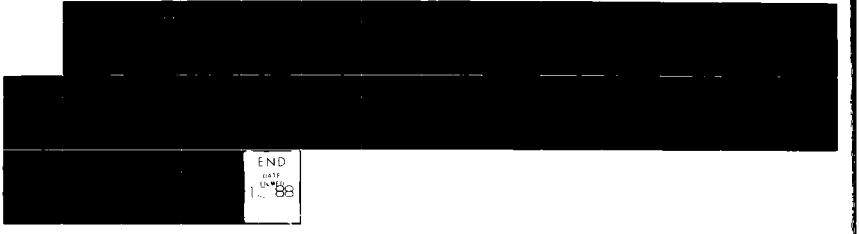
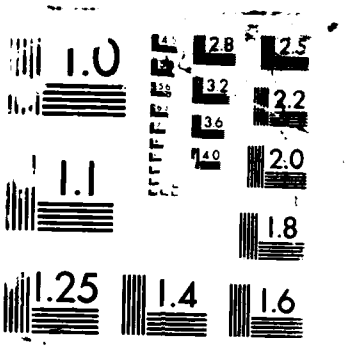
AD-A195 093 PARTITIONED STORAGE FOR TEMPORAL DATABASES(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE I AHN ET AL. 20 JAN 88 UNC-TEMPIS-19 N00014-86-K-0680 1/1

UNCLASSIFIED F/G 12/6 NL



END
DATE
6-88


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

Partitioned Storage for Temporal Databases

The TEMPIS Project

Ilsoo Ahn[†] and *Richard Snodgrass*

TEMPIS-Document No. 19

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

20 January 1988

N00014-86-K-0680

Abstract

Efficiently maintaining history data on line together with current data is difficult. This paper discusses one promising approach, the *temporally partitioned store*. The *current store* contains current data and possibly some history data, while the *history store* holds the rest of the data. The two stores can utilize different storage formats, and even different storage media, depending on the individual data characteristics. We discuss various issues on the temporally partitioned store, investigate several formats for the history store, and evaluate their performance on a set of sample queries.

1. Introduction

Databases model the real world, which is constantly changing. But conventional databases lack the capability to record and process the dynamic aspects of the changing world. They store only the latest snapshot of the enterprise being modeled. If something changes, update is made in place destroying the existing information. Using these *snapshot* databases, one cannot inquire about past information, nor perform trend analysis over a sequence of history data. Retroactive changes, error correction, audit trail, or version management must all be supported by applications in an ad-hoc manner.

If temporal support is supported in a database management systems, users can make *historical queries* to ask the status of an enterprise valid at a past or even future moment, or perform *rollback operations* to shift the reference point back in time and access the state of a database in the past. These capabilities help us understand the dynamic process of state evolution in an enterprise, and identify temporal or causal relationships among events or entities.

Steady progress in the disk technology, both magnetic and optical, continues to make larger and larger storage capacity available at a lower cost [Copeland 1982]. Hence there has been a growing interest in database systems with temporal support or version management. Bibliographical surveys, however, show that most effort has focussed on conceptual aspects such as modeling, query languages, and the semantics of time [Bolour et al. 1982, McKenzie 1986]. Little has been written on issues concerning the implementation of temporal databases.

This paper investigates new access methods tailored to the particular characteristics of database management systems with temporal support. We review previous work in Section 2, and discuss the characteristics of temporal databases that motivate the needs for the new access methods in Section 3. In Section 4, we discuss various issues of a *temporally partitioned store* that divides data into separate storage areas based on the time attribute. Section 5 presents various formats for the temporally partitioned store, and Section 6 discusses issues on secondary indexing for temporal databases. In Section 7, we evaluate and compare the performance of various access methods on a set of sample queries.

2. Previous Work

There has been a massive amount of research on the design and analysis of specific file structures with various characteristics. The resulting access methods can be categorized as either static or dynamic, depending on how they function as data accumulates.

---

† Present address of this author is AT&T Bell Laboratories, Columbus, OH 43213.

Access methods such as sequential, hashing, indexing, and ISAM are *static* in the sense that they do not accommodate growth of files without significant loss in performance. Accessing data in a sequential file requires sequential scanning, which is often too expensive. Access methods such as hashing and ISAM also suffer from rapid degradation in performance due to ever-growing overflow chains caused not only by key collisions but also by the existence of multiple versions for a single key, as will be demonstrated in Section 7. Reorganization does not help to shorten overflow chains, because all versions of a tuple share the same key. Hence performance will deteriorate rapidly not only for temporal queries but also for non-temporal queries [Ahn & Snodgrass 1986].

Dynamic access methods, such as B-trees [Bayer & McCreight 1972], virtual hashing [Litwin 1978], linear hashing [Litwin 1980], dynamic hashing [Larson 1978], extendible hashing [Fagin et al. 1979], K-D-B trees [Robinson 1981], or grid files [Nievergelt et al. 1984], adapt to dynamic growth better. These methods maintain certain structures as records are added or deleted. But the performance is still proportional on the count of all versions, which is significantly higher than the count of current versions. Furthermore, a large number of versions for some tuples will require more than a bucket for a single key, causing similar problems to those exhibited in conventional hashing. It is also difficult to maintain secondary indices for these methods, because they often split a bucket and rearrange its records. Performance problems of conventional access methods in the environment of databases with temporal support will be further discussed in Section 7.

Secondary storage cost has been decreasing steadily, and various new technologies are emerging in recent years. In particular, optical disks are becoming commercially available from several manufacturers at a reasonable cost [Fujitani 1984, Hoagland 1985]. Though optical disks provide enormous capacity at a low cost, one limitation is that they are currently write-once, not allowing reorganization or rewriting of data once they are stored. This peculiarity makes many of the conventional storage structures, especially the dynamic ones such as B-trees or dynamic hashing, unsuitable for optical disks, and requires new storage structures to utilize their potential benefits [Christodoulakis 1987].

Storage structures for temporal databases have been the topic of only a few papers. Ben-Zvi introduced the concept of the temporally-partitioned store and briefly examined reverse chaining (discussed below), future chaining (for proactive data, also discussed below), and secondary indexing of the current and history stores [Ben-Zvi 1982]. Lum, et al. also advocated reverse chaining and multiple indices, and delved further into the related topics of schema evolution, space reclamation, and index maintenance [Lum et al. 1984]. In later papers, they considered support for transaction time [Dadam et al. 1984] and integrated support for text, temporal data, and nested relations [Lum et al. 1985]. Finally, Katz and Lehman proposed a temporally-partitioned store for VLSI design files [Katz & Lehman 1984]. However, none has enumerated the possible variations of a temporally partitioned store, nor analyzed its performance.

3. Characteristics of Temporal Databases

The term *temporal database* in a generic sense refers to a database with some support for processing temporal or time-dependent data. Examples are a personnel database with a history of employee records, or an engineering database with a collection of design versions. We have identified three kinds of time to be supported in databases with different semantics, *valid time*, *transaction time*, and *user-defined time* [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. Valid time is the time when an event occurs in a real world. Transaction time is the time when a transaction occurs in a database to record the event. User-defined time is defined by a user, and its semantics is up to each application. Depending on the capability to support transaction time and valid time, we have four types of databases: *snapshot*, *rollback*, *historical*, and the *temporal database* in a narrower sense. Each of the four types has different capabilities, and faces different issues for implementation.

For the purpose of discussion, we assume that, when transaction time is supported, each datum is associated with two timestamps: the transaction identifier (an integer assigned at commit time) specifying when the datum was stored in the database (*transaction start*) and the transaction identifier specifying when the datum was logically removed from the database (*transaction stop*). When valid time is supported, each datum is associated with two timestamps: the time when the datum began to be valid in reality (*valid from*) and the time when the datum ceased to be valid in reality (*valid to*). We do not consider user-defined time. We do not assume either tuple or attribute timestamping; most of the storage methods presented below are relevant to both, though the performance ramifications of one over the other may be substantial [Ahn 1986A].

Temporal databases follow the *non-deletion* policy to preserve past information needed for historical queries or rollback operations. No record will ever be deleted once it is inserted. For each update operation, a new version is created without destroying or over-writing existing ones. This strategy solves many of the problems caused by

the *update-in-place* practice common in conventional DBMS's [Schueler 1977], but also introduces several new problems.

An immediate concern is the large volume of data to be maintained on line. Storage requirements will increase monotonically, potentially to an enormous amount, no matter what data compression technique is utilized. This problem is one of the major reasons why databases with temporal support have not been put into practice even though their benefits have been long recognized. Mechanisms must be devised to effectively deal with the ever-growing storage size, and to represent temporal versions into physical storage in such a way that past states of a database can be maintained with little redundancy.

The large amount of data to be maintained also causes long delays in accessing information. For example, the number of block accesses to get a record from an unordered file with m blocks is $O(m)$. Storing temporal data in such a file will require a large number of blocks, significantly degrading the performance. Unless temporal information is utilized as part of a key, there will be multiple versions for each single key value. However, time attributes are in general not suitable to be used as a key for storing and accessing records, for several reasons. A time attribute alone cannot be used as a key in most applications. Including time attributes in a key results in a multi-attribute key, which complicates the maintenance of the key. Even though time attributes are maintained as a part of a key, it is difficult to formulate point queries (also termed exact match queries), especially when the granularity of time values is fine. Thus, we should be able to support a range query on time attributes, which is not possible with many access methods, such as various forms of hashing.

On the other hand, there are several interesting characteristics unique to databases with temporal support. There are two distinct types of data, *current data* and *history data*, which exhibit clear differences. There is only one current version for each tuple at one time, yet multiple versions exist for some tuples in history data. Storage requirements for history data may be potentially enormous, while the size of current data is relatively static once it has stabilized. Unlike current data, history data need not be updated except when errors are corrected in the case of historical databases, which makes write-once optical disks attractive as the storage media.

There is also a correlation between the age of data and their access frequency. History data are needed less urgently than current data. While retaining history data for temporal support will encourage new applications to process history data along with current data, we still expect that new applications will be dominated by manipulation of current data. It is a challenge to exploit these unique characteristics to achieve better performance.

Considering these characteristics, conventional access methods such as hashing or ISAM are not expected to be effective for such databases with a large number of temporal versions, as was shown in an earlier study [Ahn & Snodgrass 1986]. Therefore, new access methods and storage structures tailored to the particular characteristics of database management systems with temporal support need to be developed to provide fast response for a wide range of temporal queries without penalizing conventional non-temporal queries.

The solution investigated in this paper is the *temporally partitioned store* that divides current data and history data into two storage areas. The following sections address the design decisions implied by this structure, then discuss the details of various formats for the history store. Relative advantages and disadvantages of the various formats are evaluated to determine the cost of supporting temporal queries. Issues on how to support secondary indexing for the temporally partitioned storage structure are also discussed.

4. Issues for Temporally Partitioned Store

The temporally partitioned storage structure has two storage areas, the *current store* and the *history store*. The current store contains current versions which can satisfy all non-temporal queries, and possibly some of frequently accessed history versions. The history store holds the remaining history versions. Separating current data from the bulk of history data can minimize the overhead for conventional non-temporal queries, and at the same time provide a fast access path for temporal queries. It is possible to use different access methods for each store. The current store may utilize any conventional access method suitable for a snapshot relation, such as hashing, ISAM, or B-tree. The history store may also use any conventional access method, but several variations are conceivable to exploit the concept of version inherent in history data. It is even possible to use different types of storage media; for example, history data may be stored on optical disks, while current data are kept on magnetic disks.

The temporally partitioned storage structure can also be regarded as the *reverse differential file* [Lum et al. 1985]. The scheme of differential file represents two versions of data with the main file and the differential file [Severance 1976]. The main file contains the reference version (R), and is never modified. All changes to the main file are recorded in the differential file, which are either additions (A) or deletions (D). Thus, the current version

(C) can be found by $\mathbf{R} \cup \mathbf{A} - \mathbf{D}$. Note that accessing the current version is slower than accessing the old version. On the other hand, the scheme of *reverse differential file* directly represents the current version in the file C. It also records additions (A) and deletions (D) to and from a reference version in a separate file. Then, the current version is readily available from C, and the reference version (R) can be found by $\mathbf{C} \cup \mathbf{D} - \mathbf{A}$. Since $\mathbf{A} \subseteq \mathbf{C}$, A need not be stored separately. They can, instead, be represented as a part of C by marking them with appropriate information, *e.g.* attaching time attributes to each record to show when it was appended. Attaching time attributes to each record also generalizes the number of versions from two to any number.

There are many issues to be investigated about the temporally partitioned storage structure. This section discusses criteria for splitting data between the current and the history store, update procedures for each type of databases with temporal support, methods to handle retroactive changes, proactive changes, and key changes, and the performance with regard to the update count.

4.1. Split Criteria

The main objective of the temporally partitioned storage structure in this paper is to separate current data from history data so that the overhead for supporting temporal queries can be minimized. Hence the basic criterion is to keep current versions in the current store, and to keep history versions in the history store. All non-temporal queries can be evaluated by consulting only the current store without any interference from the bulk of history versions. This criterion appears to be quite simple, but there are many complications especially with a historical or a temporal database.

The term *current version* has different implications depending on the temporal type of databases. The *version set* identifies the versions associated with a single entity. A version set usually has a single key value for all of its versions, unless there have been key changes, as will be discussed in Section 4.4. For a rollback database, the current version of a version set is the version entered into the database most recently for the version set, and has an *transaction stop* attribute value of undefined ('–'). Such tuples are put into the current store, and the other tuples are put into the history store.

But determining current versions for a historical or a temporal database is complicated by *retroactive* or *proactive* changes, which will be discussed further in Section 4.3. For a historical database, the current version has the attributes *valid from* and *valid to* overlapping with the current time. For a temporal database, the current version has the attributes *valid from* and *valid to* overlapping with the current time, and an undefined *transaction stop* value. If we ignore retroactive or proactive changes for the moment, the current store keeps tuples with a *valid to* value of '∞' for a historical database, and tuples with a *valid to* value of '∞' and an undefined *transaction stop* value for a temporal database. An extension to the temporally partitioned storage structure with the current and the history stores would be to add the third store, called the *archival* store, which contains tuples with specific *transaction stop* attribute values. The archival store will be consulted only for queries as of some moment in the past.

As discussed in Section 3, current data are in general smaller in volume than history data. Thus, the current store can be more efficient than the history store in accessing data. To take advantage of this property, we can relax the basic criterion by keeping some history data, which tend to be accessed rather frequently, in the current store. In this case, care should be taken to limit the amount of history data in the current store so that the performance of non-temporal queries would not suffer from the increased size of the current store. For example, the current store may keep up to two, instead of one, most recent versions for each version set. Furthermore, deletions or proactive changes can be handled following this criterion, as will be discussed later.

It is also possible to adopt the strategy of vertical partitioning [Ceri & Pelagatti 1984] which moves some of the current versions, with relatively low access frequencies, to the history store. A special case related with this scheme is mentioned later for proactive changes. Another factor affecting the criterion is the availability of an access path to history versions, since a version in the history store needs an access path either through some index or through a corresponding version in the current store. In conclusion, the choice of an appropriate split criterion depends in part on the number of stores supported, the access patterns to subsets of the history data, the volume of current data, the access patterns to subsets of the current data, and the access paths provided in the current and history stores.

4.2. Update Procedures

Unlike snapshot databases relying on update in place, databases with temporal support update existing information in a non-destructive way, and maintain out of date information as history data. Hence the semantics of the **append**, **delete** and **replace** statements are particularly important in databases with temporal support.

Handling the **delete** statement (abbreviated as delete) and the **replace** statement (replace) is more complicated with the temporally partitioned storage structure, which divides data between the current and the history store according to a split criterion.

This section discusses the update procedures for the temporally partitioned storage structure in each type of database. We first examine the append, delete, and replace procedures for rollback relations, then consider each in turn for historical, rollback and temporal relations. For concreteness, we discuss the update procedures in terms of the constructs available in the temporal query language TQuel [Snodgrass 1987], which is an extension of the snapshot query language Quel [Held et al. 1975]. The where clause, which is common to both Quel and TQuel, selects tuples satisfying a predicate over the non-temporal attributes; the when clause selects tuples satisfying a predicate over their valid time; and the valid clause determines the period of validity of the modification. The when and valid clauses may be used only when modifying historical or temporal relations. The details of these constructs are not important to this discussion.

For a rollback relation, **append** inserts a tuple with time attributes:

transaction start ← the current transaction identifier
transaction stop ← '–'

meaning that the tuple is effective from this transaction on.

Delete finds a tuple that has a *transaction stop* value of '–' and satisfies the where predicate, then terminates it by changing the *transaction stop* attribute to the current transaction identifier. The deleted tuple has been in the current store, and may or may not be moved to the history store depending on the split criterion. Deletion or correction of past tuples, whose *transaction stop* attribute is not '–', is not allowed in a rollback relation.

According to the basic split criterion of current data on the current store and history data on the history store, deleted tuples ought be moved to the history store. This reduces the size of the current store, but it becomes necessary to provide an access path to the version set which has no current version, lest the whole history store be scanned to locate it. The path may be a separate index of deleted tuples, or a combined index involving both the current and the history store, as will be discussed in Section 6. If the basic criterion is relaxed so that the current store may hold some of history data, deleted tuples may be left in the current store. In this case, there is no need to maintain a separate access path for deleted tuples.

Replace can be described as delete followed by append in any database. In this *delete and append* scheme, the base tuple is first deleted (in the sense of non-snapshot databases) as described above, then a copy of the base tuple with some attributes changed according to the replace statement is appended. This scheme works well with conventional storage structures, and is used by the prototype described in Section 7. But the delete and append scheme is not strictly applicable to a rollback database with the temporally partitioned storage structure. The problem is that the base tuple still stays in its place, while the newer version is put into a different location. An alternative is to append into the history store a copy of the base tuple with its *transaction stop* attribute changed to the current transaction identifier, then change the base tuple according to the replace statement. This *append and change* scheme works well for a rollback database with the temporally partitioned store, and is also better than the delete and append scheme for concurrency control and error recovery in that it reduces the critical period while the base tuple is not available.

```
range of  h  is   historical_h
append historical_h
      valid from "1/1/82" to "1/1/83"
      where (h.id = 500)
delete  h
      valid from t1  to t2
      where (h.id = 500)
```

Figure 1: Append and Delete Statements

For a historical relation, append, delete, and replace statements employ the valid clause to specify the period while any of the modification statements will be in effect. A TQuel statement in Figure 1 can be regarded as having

the *update interval* $[t_1, t_2)$, effective between t_1 and t_2 (in this case, t_1 is January 1, 1982 and t_2 is the first day of 1983). If no valid clause is specified for any modification statement, the default update interval is [now, ∞), where '∞' stands for forever. Let's call an existing tuple with the same values of the data attributes the *base tuple*, and assume it has the *base interval* $[t_{vf}, t_{vt})$, effective between t_{vf} and t_{vt}, where t_{vf} and t_{vt} are the values of the *valid from* and *valid to* attributes. Since $t_1 < t_2$ and $t_{vf} < t_{vt}$, there are six possible relationships between the base interval and the update interval as shown in Figure 2.

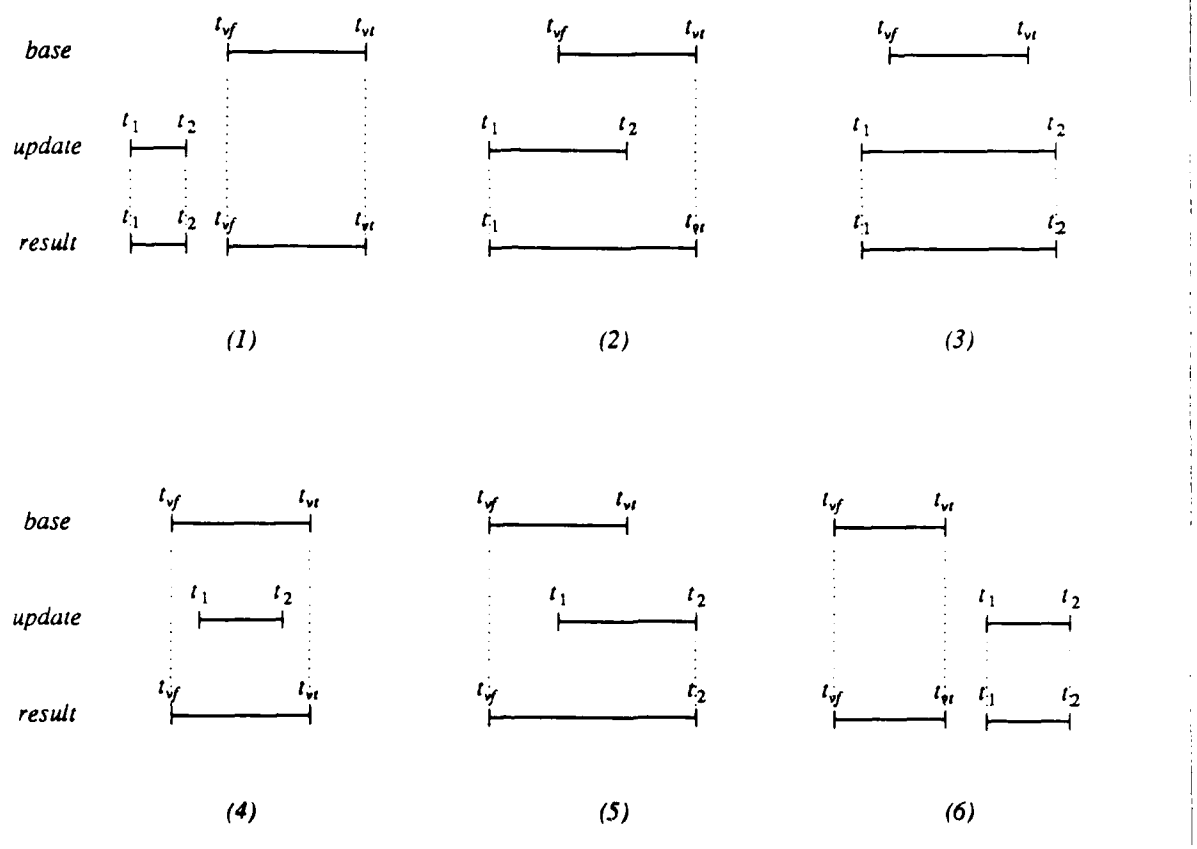


Figure 2: Base Interval *vs.* Update Interval for **append**

Append needs to be handled differently for each case.

- case (1): $t_2 < t_{vf}$
  The base interval and the update interval do not overlap, so a new tuple, valid during $[t_1, t_2)$ is simply physically appended.
- case (2): $t_1 < t_{vf} \wedge t_{vf} < t_2 \wedge t_2 < t_{vt}$
  The portion $[t_1, t_{vf})$ gets appended. The result is to change the *valid from* attribute of the base tuple to t_1. The base tuple still stays in its place, whether it is in the current or the history store.
- case (3): $t_1 < t_{vf} \wedge t_{vt} < t_2$
  Two portions, $[t_1, t_{vf})$ and $[t_{vt}, t_2)$ get appended. The result is to change the *valid from* attribute of the base tuple to t_1 and the *valid to* attribute to t_2. If the base tuple is in the history store, it may be necessary to move it into the current store depending on the split criterion.
- case (4): $t_{vf} < t_1 \wedge t_2 < t_{vt}$
  Since the base tuple completely overlaps the appended tuple, nothing needs to be done.

• case (5): $t_{vf} < t_1 \wedge t_{vt} < t_2$

The portion $[t_{vt}, t_2)$ gets appended, which changes the *valid to* attribute of the base tuple to t_2.

• case (6): $t_{vt} < t_1$

The base interval and the update interval do not overlap, so the update tuple is simply physically appended.

Thus append in a historical database is similar to append in a snapshot database for cases (1), (4), and (6), and is similar to replace in a snapshot database for cases (2), (3), and (5). Append in a rollback database is similar to case (5), except that the time axis represents transaction time.

Delete for historical relations is equally complex, and depends on the interaction between the base and update intervals (see Figure 3). Here the base tuple is an existing tuple satisfying the where and when predicates.

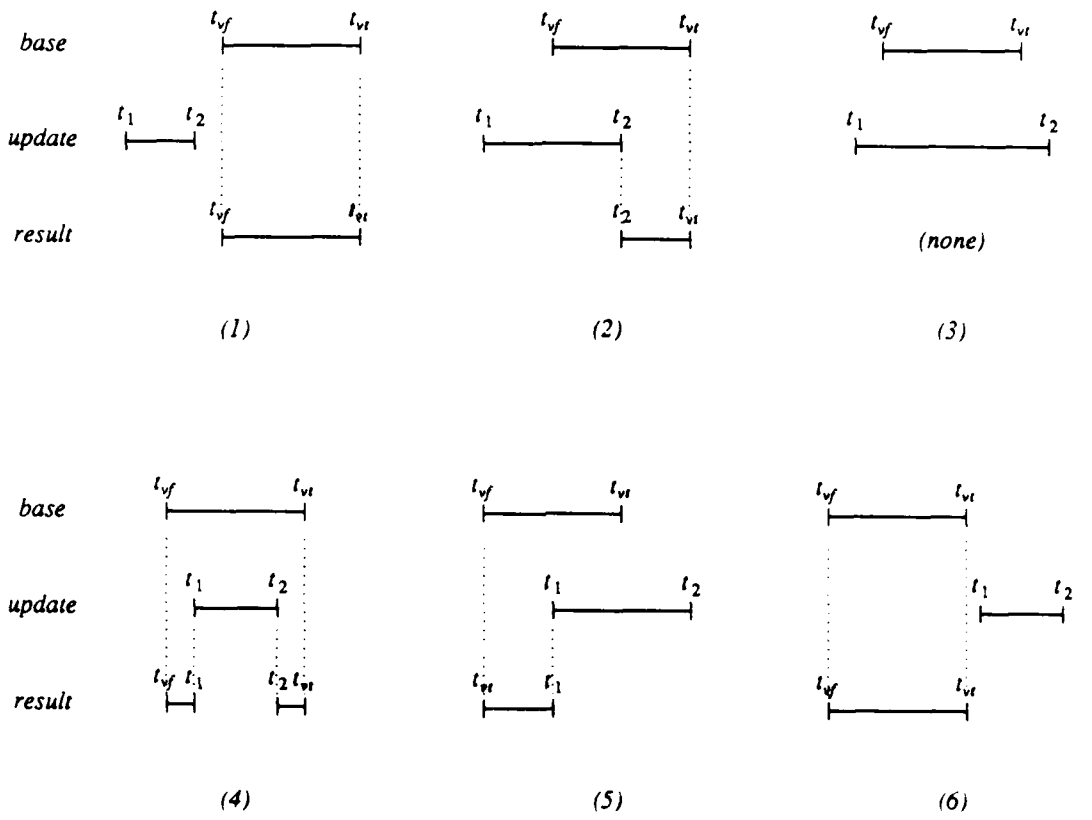


Figure 3: Base Interval *vs.* Update Interval for **`delete`**

Delete needs to be handled differently for each case, except for cases (1) and (6) which require no action.

• case (1): $t_2 < t_{vf}$

The base interval and the update interval do not overlap, so nothing needs to be done.

• case (2): $t_1 < t_{vf} \wedge t_{vf} < t_2 \wedge t_2 < t_{vt}$

The portion $[t_{vf}, t_2)$ gets deleted. The result is to change the *valid from* attribute of the base tuple to t_2. The base tuple still stays in its place, whether it is in the current or the history store.

• case (3): $t_1 < t_{vf} \wedge t_{vt} < t_2$

The base tuple is physically deleted. But the immediate predecessor version of the base tuple, if any, needs to be recognized as the most recent version of the version set in order to maintain an access path to history versions. If the base tuple is in the current store, and deleted tuples are kept in the current store, then the

immediate predecessor needs to be moved from the history store to the current store.

- case (4): $t_{vf} < t_1 \wedge t_2 < t_{vt}$
  The portion $[t_1, t_2)$, which falls on the middle of the base interval, gets deleted. First, the *valid from* attribute of the base tuple, which stays in its place, is changed to t_2. Then a new tuple, which is the same as the base tuple but with the *valid to* attribute of t_1, is inserted into the history store.
- case (5): $t_{vf} < t_1 \wedge t_{vt} < t_2$
  The portion $[t_1, t_{vt})$ gets deleted, which changes the *valid to* attribute of the base tuple to t_1. If the base tuple is in the current store, it may be necessary to move it into the history store depending on the split criterion.
- case (6): $t_{vt} < t_1$
  The base interval and the update interval do not overlap, so nothing needs to be done.

Thus delete in a historical database is similar to replace in a snapshot database, except for the case (4) which also involves a physical append, and for the cases (1) and (6) which require no action. Delete in a rollback database is similar to case (5), except that the time axis represents transaction time.

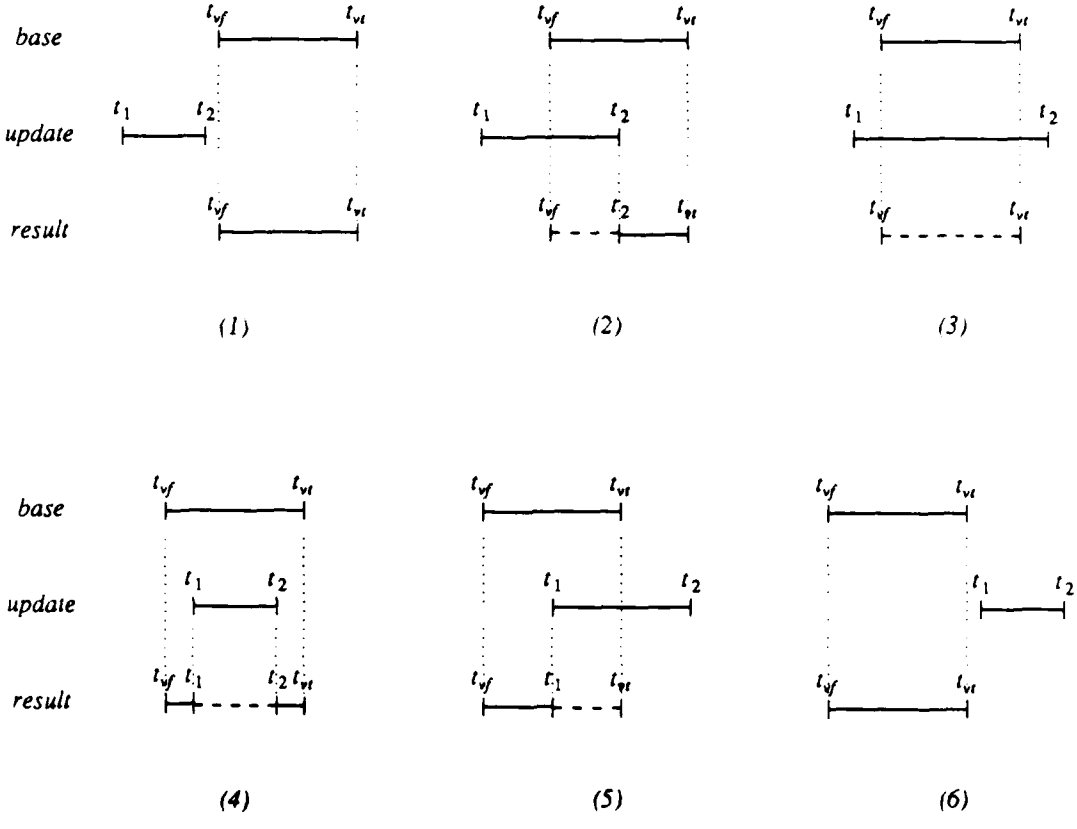


Figure 4: Base Interval *vs.* Update Interval for `replace`

Handling replace is more complicated in a historical database than in a rollback database, especially with the temporally partitioned store. To perform replace in a historical database with the temporally partitioned store, there are also six cases to be examined as shown in Figure 4, depending on the relationship between the base interval and the update interval. However, handling replace is more complicated than delete, because we need to determine the proper location of the current version and to maintain a history chain, whether explicit or not, for each version set. Basically, we follow the append and change scheme, but detailed steps vary significantly for each case.

- case (1): $t_2 < t_{vf}$
  The base interval and the update interval do not overlap, so nothing needs to be done.
- case (2): $t_1 < t_{vf} \wedge t_{vf} < t_2 \wedge t_2 < t_{vt}$
  The portion $[t_{vf}, t_2)$ gets replaced. First, the new version changed by replace is put into the history store. Its *valid from* attribute is set to t_{vf}, and its *valid to* attribute is set to t_2. Then, the base tuple gets its valid from attribute changed to t_2, but still stays in its place, whether it is in the current or the history store.
- case (3): $t_1 < t_{vf} \wedge t_{vt} < t_2$
  The new version changed by replace is put into the place of the base tuple. Its *valid from* attribute is set to t_{vf}, and its *valid to* attribute is set to t_{vt}.
- case (4): $t_{vf} < t_1 \wedge t_2 < t_{vt}$
  The portion $[t_1, t_2)$, which falls on the middle of the base interval, gets replaced. First, the new version changed by replace is put into the history store. Next, a copy of the base tuple is inserted into the history store with the *valid to* attribute set to t_1. Then, the base tuple gets its *valid from* attribute changed to t_2, but still stays in its place, whether it is in the current or the history store.
- case (5): $t_{vf} < t_1 \wedge t_{vt} < t_2$
  The portion $[t_1, t_{vt})$ gets replaced. First, a copy of the base tuple is inserted into the history store with the *valid to* attribute set to t_1. Then, the new version changed by replace is put into the place of the base tuple, whether it is in the current or the history store, with t_{vt} as the value of its *valid to* attribute. This case is particularly troublesome to the conventional delete and append scheme, because the base tuple needs to be moved to the history store. This case also corresponds to the only case for a rollback database, except that the time axis for the rollback database represents transaction time. Note that this corresponds to the case of the default valid clause for a historical database.
- case (6): $t_{vt} < t_1$
  The base interval and the update interval do not overlap, so nothing needs to be done.

Case (5) also corresponds to a replace for a rollback database, except that the time axis represents transaction time.

Though a temporal database supports transaction time in addition to valid time, modification statements for a temporal database have the same format as those for a historical database. Since the as of clause is not allowed in modification statements, transaction time does not participate in append, delete, or replace, except that the *transaction stop* attribute of the base tuple to be deleted or replaced should have the value of '–'. There are also six possible relationships between the base interval and the update interval in terms of valid time, as shown in Figures 2 through 4. For each case, append, delete and replace for a temporal database are handled in a similar manner to those for a historical database, but with two exceptions. First, a copy of the base tuple is inserted into the history store with the *transaction stop* attribute set to the current time, before the base tuple is changed in any manner. This results in adding up to three versions for each replace, but provides the capability to capture the history of retroactive and proactive changes completely, as described in the next section. Second, any tuple inserted in the process, except for the copy of the base tuple mentioned above, has the *transaction start* and *transaction stop* attributes set to the current time and '–', respectively. In addition, we need to maintain a chain of history versions for each version set, which is further complicated by the fact that each replace in a temporal database inserts at least two versions. We order versions affected in each update in reverse order of *valid from* time, then, for those versions with the same *valid from* time, in reverse order of *transaction start* time. This ordering allows us to retrieve recent versions more quickly, especially for queries with the default clause **as of** "now".

4.3. Retroactive or Proactive Changes

For a rollback database, each change is effective from the moment of the transaction, but not so for a historical or a temporal database with the valid clause. In the delete statement in Figure 1 for a historical or a temporal database, if t_1 is earlier than the current time, the change is a *retroactive from* change, and if t_2 is earlier than the current time, the change is a *retroactive to* change. If t_1 is later than the current time, the change is a *proactive from* change, and if t_2 is not '∞' but later than the current time, the change is a *proactive to* change. Thus a change may be *retroactive from* and *proactive to* at the same time, e.g., the query in Figure 1 if executed, say, in June, 1982.

Retroactive changes deal with both current and past versions, and can be handled by following the steps outlined for each case of the delete and replace statements in the previous section. However, proactive changes may involve future versions or versions to be expired which require special treatment for the temporally partitioned store.

For a proactive from change, the base tuple is still current for the moment, but will expire in time. Proactive from append or replace introduces a future version which will become current some time later. Proactive to replace introduces both a future version and a version to be expired. A question is how to handle future versions and versions to be expired. It is possible but expensive to maintain a separate index for future versions, and to monitor constantly which versions are becoming current or expired. An alternative is to keep future versions and versions to be expired together with current versions in the current store. When any of those versions is accessed in the course of query processing, it is possible to determine if it has changed its status from future to current or from current to expired, then move the expired version to the history store.

4.4. Key Changes

A *key* of a relation is a smallest set of attributes whose values uniquely identify a tuple. Formally, a key of a *snapshot relation* r over scheme R is defined as a subset K of R such that for any distinct tuples t_1 and t_2 in r, $t_1(K) \neq t_2(K)$, and no proper subset of K has this property [Maier 1985]. Thus a relation in a conventional snapshot database should not hold two tuples that agree on all the attributes of the key. However, databases with temporal support, which maintain a sequence of versions for each entity, can contain multiple tuples that agree on all the attributes of the key. Hence, the definition of the key needs to be extended for databases with temporal support.

A *key* of a relation r over scheme R in databases with temporal support is a subset K of R such that for any distinct tuples t_1 and t_2 *overlapping in time* in r, $t_1(k) \neq t_2(K)$, and no proper subset of K has this property. Two tuples t_1 and t_2 *overlap in time* if:

- for a rollback relation

  $t_1[\textit{transaction start}] \leq t_2[\textit{transaction stop}]$ $\wedge$
  $t_2[\textit{transaction start}] \leq t_1[\textit{transaction stop}]$

- for a historical relation

  $t_1[\textit{valid from}] \leq t_2[\textit{valid to}]$ $\wedge$
  $t_2[\textit{valid from}] \leq t_1[\textit{valid to}]$

- for a temporal relation

  $t_1[\textit{valid from}] \leq t_2[\textit{valid to}]$ $\wedge$
  $t_2[\textit{valid from}] \leq t_1[\textit{valid to}]$ $\wedge$
  $t_1[\textit{transaction start}] \leq t_2[\textit{transaction stop}]$ $\wedge$
  $t_2[\textit{transaction start}] \leq t_1[\textit{transaction stop}]$

The **create** data definition statement in both Quel and TQuel does not enforce the concept of the key, in that it does not specify what attributes constitute a key for a relation. Though the formal semantics for append defined for TQuel prevents two tuples identical in all the explicit attributes from overlapping in time [Snodgrass 1987], it is still up to discretion of users to observe the key constraint that any new key value entered into a relation either through append or replace does not overlap with any existing tuple with the same key value. If append or replace does not insert a new key value overlapping with any existing tuple with the same key value, update procedures for the temporally partitioned store described in Section 4.2 ensure that there is at most one active version for each key value at any moment, and thus no two tuples with the same key value overlapping in time.

Though the key value identifying an entity is not supposed to change, there are always exceptions, which cause nasty problems in conventional databases when tracking the history of changed identities. However the problem can be handled gracefully in the databases with temporal support, where a sequence of versions for each entity is maintained through physical or virtual links. If the key value of a tuple changes, a new version with the changed key becomes the current version, and the old version is kept as a history version. Thus the history of key values is captured in the same way as the history of other attribute values. But it may be necessary to rearrange the storage structure for the changed key value, if the storage structure depends on the key attributes.

4.5. Performance

A query is called *current* or *non-temporal* if it involves only current data and does not concern history data. A non-temporal query for a rollback database has the clause **as of** "now". For a historical database, a non-temporal query has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the tuple variables t_i. For a temporal database, a non-temporal query has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the tuple variables t_i, and the clause **as of** "now". Hence it is possible to

determine at compile time if a query is non-temporal.

According to the split criterion discussed in Section 4.1, all non-temporal queries can be evaluated by consulting only the current store. Therefore, maintaining history versions for temporal support does not affect the performance of conventional non-temporal queries. The only overhead is the extra space to hold implicit time attributes and possibly a physical link to history versions, which may increase the relation size and hence the cost to scan the relation.

For a temporal query, it is usually necessary to retrieve history versions from the history store. The basic algorithm accesses the current version first through the primary access path. If the temporal predicate of the query does not contain a tuple variable, we can determine the interval which satisfies the predicate. If the interval is found to be a subset of the interval denoted by the time attributes of the current version, there is no need to access the history store, because members of a version set in the history store do not overlap in time with the members of the version set in the current store. Otherwise, it is necessary to follow the chain of history versions through physical or virtual links depending on the format of the history store. However, many variations are conceivable for the structure of the history store, which greatly affects the performance of temporal queries. We can organize the history store in such a way that the cost of accessing the history store can be reduced significantly, as will be discussed next.

5. Structures of the History Store

The algorithms and the performance for accessing or updating relations with the temporally partitioned store vary significantly depending on the format of the history store. This section investigates various forms of the history store which can enhance the performance for various types of temporal queries, and analyzes their characteristics. Note that some formats can be combined together, though each format is discussed here individually. Relative advantages and disadvantages of the various formats are evaluated to determine the cost of supporting temporal queries. Section 7 analyzes the performance of these structures on a benchmark database.

5.1. Reverse Chaining

If history data are stored as a heap without any secondary access mechanism, each request for a history version must scan the whole store, which is often impractical. One solution is *reverse chaining*: all history versions of each version set are linked in reverse order starting with the current version (see Figure 5). Once the current version is located in the current store, its predecessors can be retrieved without scanning the whole history store. Each retrieval may or may not require a disk access, depending on how the versions are placed on disk. This storage structure was introduced by Ben-Zvi [Ben-Zvi 1982] and further developed by Lum, et al. [Lum et al. 1984, Lum et al. 1985].

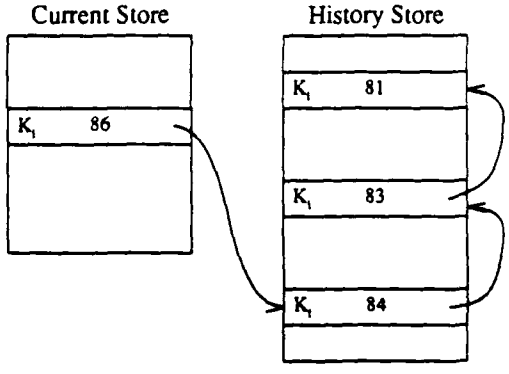


Figure 5: Reverse Chaining

For this purpose, each tuple is augmented with an extra field *nvp* (next version pointer). When a tuple is first inserted into a relation, it is put into the current store with the *nvp* field value of *null*. When a tuple is replaced, the

version existing in the current store is moved to some other place as described in Section 4.2, then a new version is put into its place with the *nvp* field containing a tuple identifier referencing the predecessor just moved. This scheme maintains the chain from the most recent to the oldest, and does not change any of existing versions in the history store, except for error correction in historical databases. Since the history store in this scheme works in an append-only mode, it can use write-once media like optical disks. If it is possible to identify attributes which will remain unchanged, *e.g.* keys; those attributes may be excluded from history versions to save space. Of course, key changes would cause complications in that case.

For a retrieve operation, the current version is located using any access mechanism available for the current store. If the query is temporal, the *nvp* field is examined. If the pointer is null or the query is non-temporal, there is no need to go through the history store. Otherwise, all its predecessors can be found by following the chain of pointers, until a version with the *nvp* of null is reached.

If the interval represented by the temporal predicate can be evaluated as constant, then the performance can be improved by exploiting the fact that all versions are ordered in the reverse order. Instead of following the chain to the end, we can stop traversing history versions when a history version is retrieved whose interval denoted by its time attributes exceeds the constant interval specified by the temporal predicate.

The lower bound for the number of block accesses to perform retrieve is $\frac{n}{b}$, when there are n history versions to be retrieved and b is the blocking factor of the history store. This occurs when all history versions are clustered together in the minimum number of blocks. The upper bound for the same case is n, when no two versions are on the same block.

When a single version set with n history versions is retrieved, the average number of block accesses, assuming uniform distribution, can be evaluated by the formula given by [Yao 1977].

$$\textit{Average Block Accesses}\,(n, f, b) = \frac{f}{b}\left[1 - \frac{\binom{f-b}{n}}{\binom{f}{n}}\right] = \frac{f}{b}\left[1 - \prod_{i=0}^{n-1}\frac{f-b-i}{f-i}\right]$$

where f is the number of records in the history store, and b is the number of records in a block of the history store. Note that reverse chaining maintains an ordering among versions belonging to the same version set, so there is no need to access a block more than once while scanning a chain of versions for a version set.

When several version sets are retrieved to process a query, the procedure to access a chain of versions is repeated for each version set. In this case, a block which contains versions belonging to several version sets may be accessed more than once. Hence the number of block accesses can exceed $\frac{f}{b}$, which is the cost to scan the history store sequentially. Let's assume that each version set has m versions, and that v version sets are retrieved. >From the formula above, it is possible determine the breakeven point when repeated traversal of history chains is still better than scanning the history store.

$$v' \times \frac{f}{b}\left[1 - \prod_{i=0}^{m-1}\frac{f-b-i}{f-i}\right] \le \frac{f}{b}$$

Thus the number of version sets v' to favor repeated traversal of history chains can be calculated numerically for a given m, the number of versions for each version set.

5.2. Accession Lists

If the length of the chain grows long in reverse chaining, it may be too slow to traverse the chain, even when only a small portion of the history versions are of interest. An alternative is to maintain *accession lists* between the current store and the history store, as shown in Figure 6.

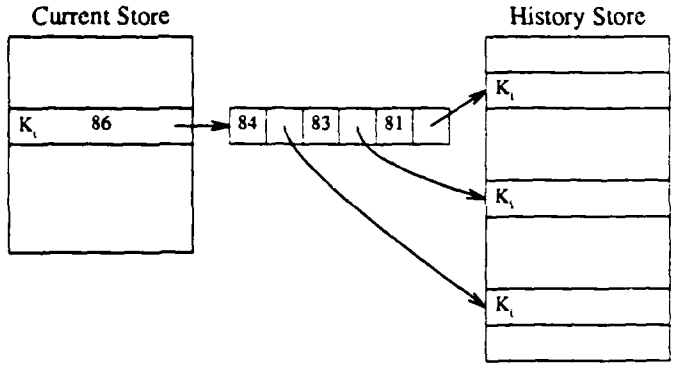


Figure 6: Accession List

A tuple is first entered into the current store, with an extra field *alp* (accession list pointer) of null. When a new version replaces the current version, the new version is put into the current store with the *alp* field pointing to an accession list, which is initialized to point to the history version just inserted into the history store. If another version is added into the version set, an entry corresponding to the version is also added into the accession list. Thus the accession list is a full index to history versions for each version set.

It is desirable to include some temporal information for each entry in accession lists, so that temporal predicates can be evaluated without actually accessing history versions. Deciding on the amount of temporal information to be included into accession lists is the classic space time tradeoff.

For a rollback relation, accession lists may contain both the *transaction start* and *transaction stop* attributes (termed a *full accession list*). Space can be saved by storing only the *transaction start* attribute (termed a *partial accession list*) without significant loss of performance, because most version sets are *contiguous*, meaning that the value of the transaction stop attribute is the same as the value of the transaction start attribute of its successor. Similar arguments apply to a historical relation, considering the *valid from* and *valid to* attributes.

For a temporal relation, accession lists may contain up to four time attributes, or some subset of the four attributes, for each version. If two time attributes are included, the *valid from* and *transaction start* attributes are recommended for the reason of contiguity mentioned above. If only one time attribute is included, the *valid from* attribute is preferable, assuming that the selectivity of the when clause is smaller than that of the as of clause, which is often the case.

For *full* accession lists, only those versions that satisfy the given temporal constraints need to be retrieved from the history store. For *partial* accession lists, it is not possible to evaluate the temporal constraints completely. Hence, all versions which *can* satisfy the constraints based on the partial information are retrieved from the history store to resolve the missing information. Still, the ratio of false hits can be significantly reduced compared with the case of no temporal information in accession lists.

Ordering of history versions in accession lists is less critical than reverse chaining, but we still recommend that they be kept in such an order that allows recent versions to be accessed more easily. Hence for a rollback database, versions are maintained in reverse order of transaction start time. For a rollback database, versions are maintained in reverse order of valid from time. For a temporal database, versions are maintained in reverse order of valid from time, then in reverse order of transaction start time.

Including temporal information in accession lists is not an overhead, as it may appear to be. When some time attributes are stored in accession lists as described above, it is not necessary to store the same information in the history store. History versions do not need an extra *nvp* field, as in reverse chaining. Accession lists are also useful to handle future versions resulting from proactive changes. The future version may be put either in the current or the history store, pointed to by an entry with appropriate temporal information in accession lists.

Since accession lists are accessed more frequently than history versions, and may be clustered or reorganized for performance reasons, it is better to keep them on magnetic disks. History versions are append only, so they may be stored on optical disks.

The upper bound for the number of block accesses to retrieve all n records is one bigger than that of reverse chaining, owing to an extra disk access for accession lists. Since temporal predicates can be evaluated without accessing the history store, the lower bound for the number of block accesses is just two including a block access for an accession list. On the average, the number of history versions actually retrieved will be much smaller than reverse chaining; determining the access cost for arbitrary queries is difficult due to the variety of temporal predicates.

5.3. Clustering

One problem with the schemes discussed thus far is that history versions belonging to a version set are scattered over several blocks. A solution is to *cluster* all versions of each version set into the minimum number of blocks (see Figure 7). Clustering significantly reduces the number of disk accesses to retrieve history versions, and thereby improves the performance of temporal queries. However, maintaining clustering while achieving a high degree of storage utilization is difficult. Clustering can be combined with other schemes described earlier, such as reverse chaining, accession lists, or indexing. Since this scheme requires splitting of blocks when overflow occurs, it is not strictly applicable to optical disks.

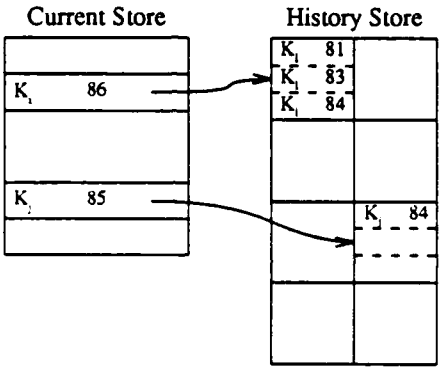


Figure 7: Clustering

There are many variations on this scheme. The simplest method is to assign a whole block to each version set with history versions. This method is a special case of *cellular chaining* to be described later, where a cell is a whole block. Unfortunately, allocating an entire block results in unacceptably low storage utilization in most cases. A better method is to share the same block for history versions of several version sets. When an overflow occurs, the block is split into two, moving all versions of some selected version sets to a new block. If all versions in the overflowed block belong to one version set, a new block is added as a successor and chained to the original block. In this scheme, $\frac{n}{b}$ blocks need to be accessed to retrieve n history versions, where b is the number of records in a block.

In the temporally partitioned storage structure, there needs to be a link between the current version and its history versions to avoid scanning the whole history store. The link may be either physical or virtual. A *physical link* is a pointer physically stored as an implicit attribute of the current version. If some history versions are moved to another location as a result of a block overflow, physical pointers in the current store pointing to those versions need to be adjusted accordingly. It is better to move the version set that has caused the overflow in this case, because it is easier to identify the version in the current store which corresponds to the versions being moved in the history store. If it is still necessary to move or compact other versions remaining in the original block, history versions need to maintain back pointers to the corresponding versions in the current store to adjust their pointers.

A *virtual link* is a conceptual link implied by some structural information. For example, history versions can be hashed on the primary key so that all versions belonging to a version set are put into one block or its overflow blocks. But the performance of conventional hashing with reasonable storage utilization deteriorates rapidly, as will

be discussed further in Section 7, if there are excessive key collisions causing long overflow chains.

One way to resolve this problem is to introduce a *scatter table* between the current store and the history store, which can serve as a combination of the physical link and the virtual link [Morris 1968]. A scatter table may have the form of an index or a directory. Each entry in a scatter table corresponds to a value hashed from the primary key of tuples in the current store, and holds a pointer to a block in the history store. When an overflow occurs to a block in the history store, the block is split into two according to a hash function which generates a sequence of different values for each occurrence of overflows. Then a new entry pointing to the new block is added to a scatter table. A scatter table plays a similar role to accession lists, but an entry in a scatter table is shared by several synonymous tuples through a hash function, while an accession list is associated with each tuple om the current store through a physical link.

Actual implementation of this scheme using a scatter table may adopt one of variable size hashing methods based on an index or a directory which can accommodate dynamic growth of a file by splitting a block upon overflow. Examples of such methods are dynamic hashing, extendible hashing, and grid files, where an index or a directory can be regarded as a scatter table described above. All three methods make it possible to retrieve a record at the cost of one block access by locating the index or directory entry for a given key, assuming that the index or the directory is small enough to reside in the main memory. If the index or the directory does not fit into the main memory, one additional disk access is necessary.

There are other variable size hashing methods which can accommodate dynamic growth of the file without maintaining an index or a directory. They are virtual hashing, linear hashing, and modified dynamic hashing. A new method of hashing termed *adaptive hashing* has been also developed [Ahn 1986B, Ahn 1987]. It can retrieve records at the cost of exactly *one* block access, even when the file size grows or shrinks dynamically. It maintains a list of overflow addresses, called the *overflow list*. Since the overflow list stores an address only when an overflow occurs, it is smaller than a directory or an index which maintains the addresses of all the buckets, and is expected to fit into the main memory. If the size of the overflow list grows too big, it is possible to reduce its size by reorganization.

5.4. Stacking

Stacking is a two dimensional implementation of a conceptual cube where all the version sets have an equal number of versions. This is useful when we are interested in the fixed number of most recent versions, where updates are rather periodic and uniformly distributed. For example, Postgres stores history data, but discards data older than a specified amount of time [Stonebraker & Rowe 1986].

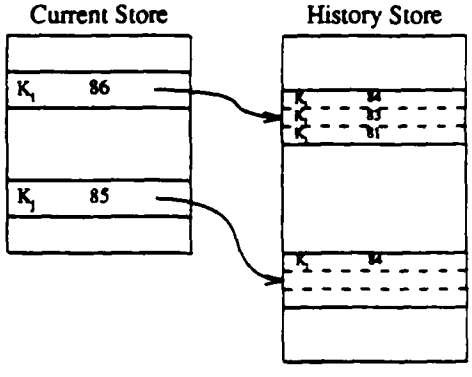


Figure 8: Stacking (depth $d = 3$)

When the first history version is put into the history store for a version set, space for d versions is allocated, where d is termed the *depth* of stacking (see Figure 8). Subsequent versions are put into the remaining portion of the allocated space. After the predetermined limit d to the number of versions is reached, the next version is put into the place of the oldest version, which becomes lost as if being pushed through the bottom of a stack.

Since the number of history versions to be maintained is predetermined, it is simple to cluster all versions belonging to a version set. Thus, the number of block accesses for retrieving n history versions is just one, assuming the entire stack fits in one block. Storage utilization is $\frac{u}{d}$ with the maximum of 100%, where u is the average update count. Increasing the depth d enables a larger number of versions to be maintained, but the storage utilization can be as low as $\frac{1}{d}$. The data being replaced by newer versions may not actually be lost, but can be archived to a lower level storage. Another interesting possibility is to organize the current store as a *shallow* stack, a stack with a small d, then store overflow data into the history store which may use any of the formats discussed in this section.

5.5. Cellular Chaining

Cellular chaining is similar to reverse chaining, but attempts to improve the performance by collecting several versions into one cell. The current version initially has an extra field *nvp* (next version pointer) of null (see Figure 9). When the first version is inserted into the history store for a version set, a cell is allocated with the size of $c \geq 1$ in the history store. The *nvp* field of the current version now points to the cell, and subsequent versions will be put into the remaining space of the cell. If this space is filled up, another cell is allocated and chained to the predecessor cell. Since the history store operates in the append only mode, this scheme can use optical disks as well.

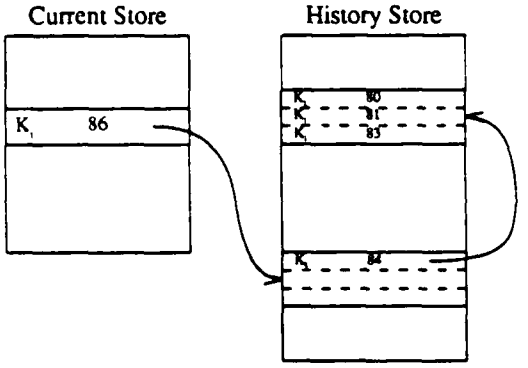


Figure 9: Cellular Chaining (cell size $c = 3$)

Cellular chaining can be regarded as a combination of reverse chaining and stacking. It also has the benefit of the clustering scheme, in that the number of blocks to be accessed is reduced by as much as a factor of c. The lower bound for the number of block accesses in retrieving n history versions is $\frac{n}{b}$, where b is the blocking factor of the history store. The upper bound is $\frac{n}{c}$, where c is the cell size of the history store. Thus increasing the cell size c improves the performance. However, a larger cell size tends to lower storage utilization. If the number of version sets are uniformly distributed, expected storage utilization can be calculated as:

$$E\ (Storage\ Utilization) = (\frac{1}{c} + \frac{2}{c} + \cdots + \frac{c}{c}) \times \frac{1}{c} = \frac{\sum_{i=1}^{c} i}{c^2} = \frac{c+1}{2c}$$

This shows that the average storage utilization is 100% for $c = 1$, which is the same as reverse chaining, ignoring the partially filled block at the end of the history store But the storage utilization falls to about 50% for a reasonably large c. It is possible to improve storage utilization by adjusting the cell size dynamically. The size of the cell can be increased linearly. For example, the first cell of each version set has the size of one, but each time a new cell is allocated for one version set, the cell size increases by one. Or the cell size may be multiplied by some factor, whenever a new cell is allocated for one version set.

6. Indexing

Performance of queries can be improved significantly by *indexing*. This section discusses the types and the structures of primary and secondary indices for databases with temporal support.

6.1. Primary Indexing

For a snapshot relation, the index is a set of <*value*, *pointer*> pairs where *value* is a key value and *pointer* is the unique identifier or the address of a tuple containing *value* as the key. For databases with temporal support, the index can be extended to include pointers to history versions. Each entry is of the form <*value*, p_c, p_{h_1}, $\cdots$ p_{h_n}>, where p_c points to the current version, and p_{h_i} with $1 \le i \le n$ points to the i-th history version.

The index entry may include temporal information so that temporal predicates can be evaluated without actually accessing data tuples. The space time tradeoff on the amount of temporal information discussed for accession lists applies in this scheme. For example, a temporal relation may have an index with a pointer and four time attributes for each version, or an index with a pointer and just one attribute, *e.g.*, *valid from*, for each version. Figure 10 illustrates this scheme, which can be regarded as a combination of conventional indexing and accession lists described above.

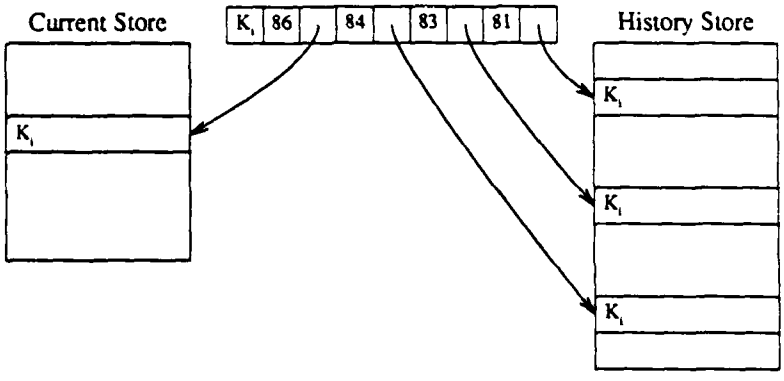


Figure 10: Indexing

Indexing is also useful to handle deleted tuples or future versions. Since history versions have an independent access path without going through the current store, all deleted tuples can be put into the history store. The future version may be put either in the current or the history store, pointed to by an index entry.

Instead of maintaining a pointer for each history version, space can be saved by storing only one entry for the list of history versions. Then each entry is of the form <*value*, p_c, p_h>, where p_c points to the current version. p_h may be the starting address of the chain of history versions, or the address of an accession list for history versions.

A generalization of this scheme is to apply the temporally partitioned structure to the index itself, maintaining two separate indices, one for the current store and the other for the history store. The benefits of the temporally partitioned store considered for storing data similarly apply to this *temporally partitioned indexing*. By separating current entries from the bulk of history entries, the current index becomes smaller and more manageable, minimizing the overhead of maintaining history versions on non-temporal queries. The history index can utilize any format developed for the history store to enhance the performance of temporal queries. For example, the current index may be hashed, while the history index has the format of accession lists. Then each entry in the current index is of the form <*value*, p_c, p_h>, as mentioned above. In any case, history versions are append-only for a rollback or a temporal relation, so they may be stored on optical disks.

Performance characteristics of the indexing scheme are similar to that of accession lists. The upper bound for the number of block accesses to retrieve all n records is n, one less than that of accession lists, without counting the cost to access the index itself. The lower bound for the number of block accesses is just one, without counting the cost to access the index itself. Since temporal predicates can be evaluated by using temporal information included

in the index, the number of history versions actually retrieved may be much smaller than reverse chaining.

One problem with indexing is that the format of the current store is tied to indexing, while other schemes allow any format for the current store. Another problem is accessing records through non-key attributes. It is necessary to maintain the same ordering for the index and the current store, so that the current store can be scanned synchronously with the index.

6.2. Secondary Indexing

For a snapshot relation, a secondary index is a set of <*value, pointer*> pairs, where *value* is a secondary key and *pointer* is the unique identifier or the address of the corresponding tuple. Since the *value* is not expected to be unique, there may be several entries for a single *value*. There will be more entries for each *value* in a secondary index for a relation with temporal support, because it maintains history versions in addition to current data. A typical query retrieves only a small subset of all the versions for a given *value*, but temporal predicates to determine which versions satisfy the query can be evaluated only after accessing the data themselves. The number of false hits can be reduced if some or all of temporal information is also maintained in a secondary index. Therefore, extension of the conventional secondary index is desirable for each type of databases with temporal support.

For a rollback database, a secondary index itself can be a rollback relation augmented with *transaction start* and *transaction stop* attributes. Then each index entry is a quadruple <*value, pointer, transaction start, transaction stop*>. There is the overhead of 8 bytes for each entry, but the as of clause can be evaluated from the information in the secondary index. Only the tuples satisfying the as of clause need to be retrieved, significantly enhancing the performance. If the version sets are contiguous or nearly contiguous, storing only the *transaction start* attribute can save space without significant loss of performance. The same argument applies to a historical database, when the valid clause is substituted for the as of clause, and the *valid from* and *valid to* attributes are used.

For a temporal database, a secondary index may be a rollback relation, a historical relation, or a temporal relation itself. If the index is a rollback relation, the as of clause can be evaluated from the information in the index. Then those versions that satisfy the as of clause are retrieved from the current or the history store to resolve the valid predicate. If the index is a historical relation, those tuples that fail the valid clause need not be accessed to resolve the as of predicate. If the index is itself a temporal relation, each index entry is a sextuple <*value, pointer, valid from, valid to, transaction start, transaction stop*>. There is the overhead of 16 bytes for each entry, but temporal predicates of the valid and the as of clauses can be evaluated completely from the information in the secondary index. It is also possible to store some subsets of the four time attributes, *e.g.*, *valid from* and *transaction start*, or only one of the two. Storing only a subset saves space, but the number of false hits will increase. The type of secondary indices available for each type of databases is summarized in the Figure 11. Deciding which type of secondary index to use for a database with temporal support is a tradeoff of space versus time.

	Snapshot	Rollback	Historical	Temporal
Snapshot Database	√			
Rollback Database	√	√		
Historical Database	√		√	
Temporal Database	√	√	√	√

Figure 11: Types of Secondary Indices for Each Type of Databases

7. Performance Evaluation

In this section, we present a preliminary performance evaluation of the access methods discussed in Sections 5 and 6. Disk read counts for sixteen queries executed on rollback, historical and temporal relations for a variety of access methods were analytically derived using a model described elsewhere [Ahn & Snodgrass 1988]. The model consists of four transformations through a series of intermediate expressions based on the characteristics of database/relations and storage devices. A temporal query is mapped to an *algebraic expression* which is transformed to a *file primitive expression*. A file primitive expression, in turn, is transformed to an *access path expression*, and finally to the access cost. Since conventional databases are subsets of temporal databases, the model can be used to analyze the performance of conventional databases as well.

To validate the model, we also ran the same queries on a prototype temporal DBMS built by extending the snapshot DBMS Ingres [Stonebraker et al. 1976]. We compared these derived disk read counts with those measured on the prototype. The results indicate that the cost, expressed as number of disk accesses, of a query on a relation supporting transaction or valid time can be estimated quite accurately, within 1% for query executions under controlled circumstances (e.g., ignoring buffering artifact) [Ahn & Snodgrass 1988].

7.1. The Prototype

The prototype supports TQuel and handles all four types of databases: snapshot, rollback, historical and temporal. It also supports append, delete, and replace statements of TQuel for the four temporal types. Each tuple of a rollback or a temporal relation is augmented with two timestamp attributes (each a 32-bit integer) for transaction time, and each tuple of a historical or temporal relations is augmented with one or two timestamp attributes (each also a 32-bit integer) for valid time.

The default storage format of a relation in INGRES, and hence in the prototype, is a *heap*. The modify statement in Quel converts the storage structure of a relation from one format to another. Major storage options available in INGRES are:

heap : for a sequential file
hash : for a hashed file
isam : for an ISAM file

For example, the statement

```
modify Temporal_h to hash on id where fillfactor = 100
```

converts the `Temporal_h` relation to a hashed file with the loading factor of 100%.

In the prototype, new options were added to the modify statement to specify the format of the history store for the temporally partitioned storage structure. They are:

chain : for *reverse chaining*
accessionlist : for *accession lists*
cluster : for *clustering*
stack : for *stacking*
cellular : for *cellular chaining*
index : for *primary indexing*

For example, the statement

```
modify Temporal_h to chain on id
```

changes the `Temporal_h` relation to the temporally partitioned store, if it is not already in such a structure. The history store uses reverse chaining with the `id` attribute as the key, while the current store maintains the previous format.

Some formats require additional parameters. Accession lists and indexing have the parameter **time** to specify the amount of temporal information to be maintained in accession lists or index entries. Allowed values for the time parameter are **all** to maintain information on all the time attributes, or a list of time attributes such as **valid from, valid to, transaction start,** and **transaction stop**. For example, we use the following statement to change the history store to the format of accession lists with all the time attributes:

```
modify Temporal_h to accessionlist on id where time = (all)
```

Stacking and cellular chaining have the parameter cellsize to specify the stacking depth or the size of a cell. To change the history store to the format of cellular chaining with up to four tuples in each cell, we use the statement:

```
modify Temporal_h to cellular on id where cellsize = 4
```

Issuing another modify statement with one of the options **heap**, **hash**, or **isam** will change the format of the current store accordingly, but the history store will be unaffected. The option **single** was also added to convert a relation from the temporally partitioned structure to the single file structure. Therefore, we can specify that the structure of a relation be changed from a single file to another single file structure, from a single file structure to a temporally partitioned store, from a temporally partitioned store to another temporally partitioned store, or from a temporally partitioned to a single file structure. In this process, we can change the formats of the current and the history store independently of each other.

We can determine at compile time if a query is non-temporal. For a rollback database, a query is non-temporal if it has the clause **as of** "now". For a historical database, a query is non-temporal if it has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the range variables t_i. For a temporal database, a query is non-temporal if it has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the range variables t_i, and the clause **as of** "now". For a non-temporal or current query, the query is evaluated by consulting only the current store without going through the history store, using the conventional access methods provided by INGRES.

For the delete or the replace statement on a rollback database, there is only one case to be examined for the relationship between the base interval and the update interval. For the delete or the replace statement on a historical or a temporal database, there are four cases to be examined, ignoring two null cases, for the relationships between the base interval and the update interval as discussed in Section 4.2.

The delete and append scheme was found to be inapplicable to the temporally partitioned store, because the base tuple remains in its place, while the newer version is put into a different location. Thus, the system was changed to follow the append and change scheme as discussed in Section 4.2. We had to examine each case of the relationships between the base interval and the update interval carefully to determine the proper location of the current version, and to maintain a history chain, whether explicit or not, for each version set. Maintaining a chain of history versions for each version set is more complicated for a temporal database, since each replace inserts at least two versions. We ordered versions affected in each update in reverse order of *valid from* time, then in reverse order of *transaction start* time. Thus, we can retrieve recent versions more quickly, especially for queries with the default clause **as of** "now".

Accessing a relation with the single file structure involves two steps: one for the main block and the other for overflow blocks. Accessing a relation with the temporally partitioned structure involves another step: following the history chain, whether explicit or implicit. Hence we need to maintain global information on which store provides the tuple being processed now and the tuple to be retrieved next. Algorithms to handle the delete and the replace statements on different types of relations are given elsewhere [Ahn 1986B].

For simplicity, the split criterion adopted in implementing the temporally partitioned store was:

- The current store contains current versions, while the history store holds history versions.
- Deleted tuples are kept in the current store.
- Versions to be expired, discussed in Section 4.3, are kept in the current store until a new version is inserted.
- Future versions are stored in the current store.

At present, the structure of reverse chaining has been implemented. The prototype's parser accepts the full BNF syntax, but the remaining components do not support the other options.

We assume that accession lists and indexing maintain complete temporal information, both *transaction time* and *valid time* as appropriate, separate from history data. The index itself is assumed to be a hashed file, but note that indexing restricts the format of the current store to indexing, as discussed in Section 5.2. We also assume that the depth for stacking is four, and the cell size for cellular chaining is four. Finally, for hashing we assume that the hash function is well-behaved (a badly-behaving hash function would create many more overflow blocks).

For clustering, we use the method of adaptive hashing. The average storage utilization for adaptive hashing is 69.3 % [Ahn 1987]. However, databases considered in this analysis have high update counts, so each version set consists of more versions than a block can hold. When a block gets full with versions belonging to a single version set, we need to maintain a chain of overflow blocks. As a result, storage utilization becomes 100% except for the last block of each chain.

7.2. The Benchmark

The benchmark queries reference two temporal relations, one hashed on the `id` attribute and one structured as an ISAM file on the `id` attribute (see Figure 12).

```
        range of h  is Temporal_h          /* hashed on id */
        range of i  is Temporal_i          /* ISAM   on id */

Q01 : retrieve (h.id, h.seq)  where h.id = 500

Q02 : retrieve (i.id, i.seq)  where i.id = 500

Q03 : retrieve (h.id, h.seq)  as of "08:00 1/1/80"

Q04 : retrieve (i.id, i.seq)  as of "08:00 1/1/80"

Q05 : retrieve (h.id, h.seq)  where h.id = 500
            when  h overlap "now"

Q06 : retrieve (i.id, i.seq)  where i.id = 500
            when  i overlap "now"

Q07 : retrieve (h.id, h.seq)  where h.amount = 69400
            when  h overlap "now"

Q08 : retrieve (i.id, i.seq)  where i.amount = 73700
            when  i overlap "now"

Q09 : retrieve (h.id, i.id, i.amount)     where h.id = i.amount
            when  h overlap i and   i overlap "now"

Q10 : retrieve (i.id, h.id, h.amount)     where i.id = h.amount
            when  h overlap i and   h overlap "now"

Q11 : retrieve (h.id, h.seq, i.id, i.seq, i.amount)
            valid from begin of h   to end of i
            when  begin of h precede i
            as of "4:00 1/1/80"

Q12 : retrieve (h.id, h.seq, i.id, i.seq, i.amount)
            valid from begin of (h overlap i)   to end of (h extend i)
            where h.id = 500  and   i.amount = 73700
            when  h overlap i
            as of "now"

Q13 : retrieve (h.id, h.seq)  where h.id = 455
            when  "1/1/82" precede  end of h

Q14 : retrieve (h.id, h.seq)  where h.amount = 10300
            when  "1/1/82" precede  end of h

Q15 : retrieve (h.id, h.seq)  where h.amount = 10300
            as of "1/1/83"

Q16 : retrieve (h.id, h.seq)  where h.amount = 10300
            when  "1/1/82" precede  end of h
            as of "1/1/83"
```

Figure 12: Benchmark Queries

With these queries, we were able to examine the following types of accesses:

- version scanning (all versions of a tuple),
- rollback queries,
- historical queries,
- key versus non-key access,
- ISAM versus hashed key access, and
- temporal and non-temporal joins.

The sixteen queries on these relations allowed us to compare 196 query executions, resulting in a fairly comprehensive evaluation of the model. We now discuss the storage and accessing costs for the various storage structures. We only consider temporal relations and assume a loading factor of 100%.

7.3. Space Requirements

Space requirements for update counts of 0 and 14 are shown in Figure 13 for the `Temporal_h` relation in hashing (the sizes for ISAM are similar), and for the same relation with various formats of the temporally partitioned structure. Space requirements for the `Temporal_i` relation are similar to the `Temporal_h` relation except that the ISAM file requires additional space for directories. The table also shows the *growth rate*, which is obtained when the *growth per update* is divided by the size for the update count of 0. In the case of stacking, the size stabilizes when an update count of 4 is reached.

Type	Hashing	Reverse Chaining	Accession Lists	Clustering	Stacking	Cellular Chaining	Indexing
Size, UC= 0	129	147	147	147	147	147	141
Size, UC=14	3717	4243	3957	4243	(733)	4243	4802
Growth per Update	256.3	292.6	272.1	292.6	(41.9)	292.6	281.5
Growth Rate	1.99	1.99	1.85	1.99	(0.28)	1.99	2.0

Notes :
'UC' denotes *Update Count*.
'(*n*)' denotes that only a partial history is stored.

Figure 13: Space Requirements for the `Temporal_h` Relation

7.4. Input Costs

From Figure 13, we can make the following observations on the storage requirements of a temporal relation with the temporally partitioned storage structure:

- The temporally partitioned storage structures consume slightly more space than the single file structure when the update count is 0, due to extra space for a physical link to the history chain.
- The temporally partitioned storage structures consume more space than the single file structure when the update count is not 0, due to extra space for maintenance of chaining, indexing or accession lists.

- When the update count is not 0, space requirements for reverse chaining, accession lists, indexing, and clustering are about the same.
- When the update count is not 0, space requirements for cellular chaining can be larger than the other formats if there are unfilled cells.
- When the update count is not 0, storage size for stacking remains the same, but older versions are lost due to stack overflows.

Figure 14 shows the input costs for the benchmark queries of Figure 12 on the temporal database; the queries and benchmark relations were crafted so that the output costs were similar for all queries. The input costs do not take into account the benefits of buffering; hence, they are probably overestimated in some cases. Two columns under the label *Conventional* show the queries costs for the update count of 0 and 14. The remaining six columns show the costs of queries for the update count of 14 for various formats of the history store. When the update count is 0, the cost for any of the temporally partitioned structures is the same as the cost for the conventional case. The figures in the first column were derived from the model then validated by executing the queries on the prototype [Ahn & Snodgrass 1988]. The figures in the remaining columns were derived from the model. Details of the analysis are available elsewhere [Ahn 1986B].

Query	Conventional		Temporally Partitioned Store for Update Count = 14					
	Update Count		Reverse Chaining	Accession Lists	Clustering	Stacking	Cellular Chaining	Indexing
	0	14						
Q01	1	29	29	30	5	(2)	8	30
Q02	2	30	31	30	6	(3)	9	30
Q03	129	3717	4243	776	4243	X	4243	787
Q04	128	3712	4243	776	4243	X	4243	787
Q05	1	29	1	1	1	1	1	2
Q06	2	30	2	2	2	2	2	2
Q07	129	3717	147	147	147	147	147	141
Q08	128	3712	147	147	147	147	147	141
Q09	1200	33350	1227	1227	1227	1227	1227	2218
Q10	2233	34493	2251	2251	2251	2251	2251	2218
Q11	385	11141	12729	2317	12729	X	12729	2350
Q12	131	3743	4274	3989	4250	(737)	4253	4114
Q13	1	29	29	8	5	X	8	8
Q14	129	3717	4243	3957	4243	X	4243	4082
Q15	129	3717	4243	3957	4243	X	4243	4082
Q16	129	3717	4243	3957	4243	X	4243	4082

Notes :

'X' denotes *not applicable.*

'(*n*)' denotes that only a partial answer is retrieved.

Figure 14: The Temporal Database with 100% Loading

The advantage of the temporally partitioned store is evident in processing current queries such as Q05 through Q10. For queries Q05 through Q10 on any temporally partitioned structure other than indexing, the cost remains constant regardless of the update count. For example, Q10 on the temporal database costs 2251 block accesses instead of 34493 block accesses when the update count is 14. Note, however, that the costs of queries Q07 through Q10 on a temporally partitioned structure are slightly higher than the corresponding costs on a conventional structure with the update count of 0, because the size of the current store is bigger than the conventional structure with the update count of 0. As for query Q09 or Q10 on the temporal database with indexing, we need to scan the index and the current store of the `Temporal_i` relation, then repeatedly access the `Temporal_h` relation through the index. The resulting cost is significantly higher than other formats, but is still lower than the conventional case.

The performance of temporal queries like Q01 and Q02 can be improved by clustering, which collects history versions of each version set into a minimum number of blocks. Cellular chaining also provides the benefit of clustering to a certain degree. Seven cells are required to hold 28 history versions assuming the cell size of four. Hence, Q01 costs eight block accesses, and Q02 costs nine block accesses.

By stacking, we can retrieve history versions for each version set at the cost of one block access, but only a limited number of the most recent versions are maintained. Thus, Q01 costs two block accesses, and Q02 costs three block accesses, but those figures are put in parentheses to denote that the answers are only partial. Note that stacking cannot answer queries Q03 and Q04 inquiring the old status of the database, because older versions of history data were discarded due to stack overflow.

Accession lists with temporal information can facilitate temporal queries Q03 and Q04 by evaluating the temporal predicate without accessing history data. Similar improvement is also achieved by indexing, where each index entry maintains complete temporal information for transaction time. We need not scan the current store in indexing, so entries satisfying the as of clause are extracted while canning the index for the `Temporal_h` relation.

As for query Q11 which requires a join operation on time attributes, the performance can be improved by accession lists, where each accession list maintains complete temporal information for all the time attributes. Since each entry with four time attributes and a pointer to a history version consumes 20 bytes, and there are 28 history versions times 1024 version sets for the update count of 14, the size of the collected accession lists is 624 blocks. By scanning the current store and the entries in the accession lists for the `Temporal_h` relation, the entries satisfying the as of clause are extracted. If we assume that the number of such entries is two, and that *tuple substitution* is used to perform a join (as in the case in Ingres and in the prototype), then the current store and the accession lists for the `Temporal_i` relation are scanned twice to find entries satisfying the as of and the when clauses. Thus we end up with scanning the current store and the accession lists three times: once for the `Temporal_h` relation and twice for the `Temporal_i` relation. If we assume that two entries in the accession lists for the `Temporal_i` relation satisfy the as of and the when clause, then four history versions are actually retrieved from the history store: two from the `Temporal_h` relation and two from the `Temporal_i` relation. So the total cost is 2317 block accesses ($= (147 + 624) \times 3 + 4$), which is a marked improvement from 11141 of the conventional method. The improvement results from performing a temporal join on the accession lists, whose size is much smaller than the history data.

Similar improvement is also achieved by indexing, where each index entry maintains complete temporal information for the four time attributes. Since each entry with four time attributes plus a key and a pointer takes 24 bytes, and there are 29 versions times 1024 version sets for the update count of 14, the size of the entire index is 782 blocks. Scanning the index for the `Temporal_h` relation, the index entries satisfying the as of clause are extracted. Under similar assumptions to the case of accession lists above, the index for the `Temporal_i` relation is scanned twice to find the entries satisfying the as of and the when clauses. Then the total cost is 2350 block accesses ($= 782 \times 3 + 4$).

Query Q12 is facilitated by clustering or cellular chaining for the portion of scanning a version set, as discussed for queries Q01 and Q02, but the overall performance is dominated by scanning the `Temporal_i` relation sequentially. Secondary indexing is necessary to improve the performance of query Q12. Note that stacking cannot answer query Q11, and provides only a partial answer to query Q12.

Query Q13 is similar to Q01, but Q13 can be improved by accession lists or indexing. The when clause can be evaluated without accessing history data, then only the tuples satisfying the temporal predicate are retrieved. If we assume there are seven such tuples, the cost is eight block accesses, where one extra block accounts for accessing an accession list or an index entry.

Queries Q14 through Q16 retrieve tuples through a non-key attribute, which requires sequential scanning of the entire relation. Maintaining a secondary index can improve the costs of these queries, as will be discussed in the next section.

7.5. Secondary Indexing

Queries retrieving records through non-key attributes can be facilitated by secondary indexing. For example, we can create a secondary index, `Temp_h_inx`, on the `amount` attribute of the `Temporal_h` relation using the **`index`** statement in Quel:

```
index on Temporal_h is Temp_h_inx (amount)
```

Maintaining a secondary index on the attribute `amount` can improve the performance of queries such as Q07,

Q08, Q12, and Q14 through Q16.

As discussed in Section 6.2, there are several types of secondary indices, especially for a temporal relation. A secondary index for a temporal relation may be any of snapshot, rollback, historical, or temporal. To specify the type of a secondary index, we extend the index statement with the **as** *type* clause, where the type can be any of **snapshot, rollback, historical**, and **temporal**. For example, a statement:

```
index on Temporal_h is Temp_h_inx (amount) as temporal
```

creates a secondary index as a temporal relation.

The default storage structure for a secondary index is a heap, but like any regular relation, its structure can be changed to other format using the modify statement. An index may be stored into a single file for all the versions (single file), or may itself be maintained as a temporally partitioned structure having a current index for current data and a history index for history data. In each case, we may choose any access methods such as a heap, hashing, ISAM, etc. At present, our prototype supports only the secondary indices as snapshots. The other options were not implemented into the prototype.

Space requirements for various types of secondary indices on the *Temporal_h* relation are shown in Figure 15, when the update count is 0 or 14. The table also shows the *growth rate*, which is obtained when the *growth per update* is divided by the size for the update count of 0. Compared with the table in Figure 13, a secondary index consumes from 8% to 21% of the space required by the relation itself.

Type	as Snapshot	as Rollback	as Historical	as Temporal
Size, UC= 0	11	19	19	27
Size, UC=14	295	531	531	782
Growth per Update	20.3	36.6	36.6	53.9
Growth Rate	1.85	1.93	1.93	2.0

Note: 'UC' denotes *Update Count*.

Figure 15: Space Requirements for a Secondary Index

For the snapshot index, each entry needs eight bytes, four for the secondary key and four for a pointer. Since a block of 1024 bytes can store 101 entries, 11 blocks are needed for 1024 tuples when the update count is 0. When the update count is 14, there are 29 versions multiplied by 1024 tuples; hence 295 blocks are needed for the single file index.

For the rollback or the historical index, each entry needs 16 bytes, four for the secondary key, four for a pointer, and eight for two attributes of transaction time or valid time. So a block of 1024 bytes can store 56 entries, and there are 29 versions multiplied by 1024 tuples when the update count is 14; hence 531 blocks are needed for the single file index.

For the temporal index, each entry needs 24 bytes, four for the secondary key, four for a pointer, eight for two attributes of valid time, and eight for two attributes of transaction time. So a block of 1024 bytes can store 38 entries, and there are 29 versions multiplied by 1024 tuples when the update count is 14; hence 782 blocks are needed for the single file index.

Figure 16 compares the snapshot index with the rollback index in terms of the costs of sample queries on the temporal database with the update count of 14. Performance figures in this table were derived analytically. The existence or the structure of secondary indices do not affect the performance of other queries which do not involve the secondary access path.

Query	Conventional		Indexed as Snapshot				Indexed as Rollback			
	Update Count		as Single		as Partitioned		as Single		as Partitioned	
	0	14	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash
Q07	129	3717	324	30	12	2	560	30	20	2
Q08	128	3712	324	30	12	2	560	30	20	2
Q12	131	3743	355	61	355	62	591	61	591	62
Q14	129	3717	324	30	324	31	560	30	560	31
Q15	129	3717	324	30	324	31	543	13	543	14
Q16	129	3717	324	30	324	31	543	13	543	14

Note: All values are for a temporal database with an update count of 14.

Figure 16: Secondary Indexing as Snapshot or Rollback

If the index is stored as a heap, queries Q07 and Q08 cost 324 block accesses each, 295 index blocks plus 29 data blocks. This is in fact more expensive than the simple temporally partitioned store without any index, though better than the conventional structure. Hence, we must take care that the cost of using an index does not overwhelm the saving obtained from using the temporally partitioned store. If the index is hashed, the cost is reduced to 30 block accesses with 1 index block and 29 data blocks.

If we follow the temporally partitioned scheme maintaining a separate index for current data, there are only 1024 entries in the current index, requiring 11 index blocks. Each of Q07 and Q08 costs 12 blocks with a heap index, while it costs only 2 blocks with hashing. Note the difference between 3717 blocks and 2 blocks for processing the same query.

Query Q12 can also benefit from secondary indexing, since it is no longer necessary to scan the `Temporal_i` relation sequentially. If the index is stored as a single heap, Q12 costs 355 block accesses, where 295 block accesses are needed to scan the index. If the index is stored as a single hash, the cost is reduced to 61 block accesses.

Queries Q14 through Q16 are similar to queries Q07 and Q08 in that they are one relation queries and their costs can be reduced significantly with secondary indexing. However, queries Q14 through Q16, like Q12, are temporal queries, and need to access history data regardless of the storage structure. Thus the temporally partitioned index is not better than the single file index for queries Q12 and Q14 through Q16. In fact, the index as a temporally partitioned hash costs one more block access than the index as a single hash, because each index needs to be hashed separately.

The rollback index is effective for processing queries with the as of clause, such as Q15 and Q16. The as of predicate can be evaluated with information from index entries, and only the tuples that satisfy the predicate need to be retrieved.

If the index is stored as a single hash, query Q15 costs 13 block accesses, assuming that there are 12 tuples satisfying the **as of** clause among 29 candidates. Storing the index as a temporally partitioned hash, query Q15 costs 14 block accesses, one block access more than as a single hash, since each index needs to be hashed separately. However, storing the rollback index as a heap increases the query costs over the snapshot index, due to the bigger size of the rollback index.

Figure 17 compares the historical index with the temporal index in terms of the costs of sample queries on the temporal database with the update count of 14. The discussion on the rollback index similarly applies to the historical index, except that the historical index maintains two attributes of valid time instead of transaction time, and that the historical index is effective for processing queries with the when clause like Q14 and Q16. If the index

is stored as a single hash, Q14 or Q16 costs 8 block accesses, assuming that there are 7 tuples satisfying the **when** clause among 29 candidates.

Query	Conventional		Indexed as Historical				Indexed as Temporal			
	Update Count		as Single		as Partitioned		as Single		as Partitioned	
	0	14	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash
Q07	129	3717	532	2	20	2	783	2	28	2
Q08	128	3712	532	2	20	2	783	2	28	2
Q12	131	3743	563	61	563	62	814	61	814	62
Q14	129	3717	538	8	538	9	789	8	789	9
Q15	129	3717	560	30	560	31	794	13	794	14
Q16	129	3717	538	8	538	9	786	5	786	6

Note: All values are for a temporal database with an update count of 14.

Figure 17: Secondary Indexing as Historical or Temporal

The temporal index combines the benefits of the rollback index and the historical index, effective for processing queries with the as of or when clause. The temporal predicate can be evaluated completely with information from index entries, and only the tuples that satisfy the predicate need to be retrieved.

If the index is stored as a single hash, Q16 costs only 5 block accesses, assuming that there are 4 tuples satisfying both the when and the as of clauses among 29 candidates. However, storing the temporal index as a heap increases the cost of queries over any other types of indices, due to the bigger size of the temporal index.

Now we can make the following observations on the types of secondary indices, based on the analysis of query costs as shown in Figures 16 and 17.

- The temporally partitioned index is good for non-temporal queries.
- For temporal queries, the cost of a query for the temporally partitioned heap index is equal to the cost of the query for the single heap index.
- For temporal queries, the cost of a query for the temporally partitioned hash index is more expensive by one block access than the cost of the query for the single hash index.
- The rollback secondary index is good for queries with the as of clause.
- The historical secondary index is good for queries with the when clause.
- The temporal secondary index is good for queries with either or both of the when and the as of clauses.
- *It is desirable to provide a secondary index with the random access mechanism such as hashing.*
- If there is no random access mechanism for a secondary index, storing a large amount of temporal information in index entries degrades the performance due to its bigger size.

8. Conclusions

Database systems with temporal support maintain history data on line together with current data, which causes problems in terms of both space and performance. This paper discussed the temporally partitioned store that could provide fast response for various temporal queries without penalizing conventional non-temporal queries. The current store holds current data and possibly some history data, while the history store contains the rest. We examined various issues concerning the temporally partitioned store, and investigated several formats for the history store, including reverse chaining, accessing lists, clustering, stacking, and cellular chaining. Storage structures for primary and secondary indices were also considered. We evaluated the performance of all of these storage structures on a set of sample queries, compared them with conventional access methods, and demonstrated significant improvement in some cases.

This work should be viewed as an initial attempt to characterize storage structures for temporal databases utilizing temporal partitioning. While the desirability of a temporally partitioned store has bee shown, the evaluation of individual storage structures and their applicability, whether to the current store, history store, or index, should be more extensively explored. Specifically, our results should be extended to a wider range of queries, to update transactions, to various combinations of attribute and tuple timestamping, to comparisons with additional implementations, to consideration of the effect of buffering and sequential access, and to consideration of the interaction with concurrency control and recovery.

9. Acknowledgements

This research was supported by NSF grant DCR-8402339. The work of the second author was also supported by an IBM Faculty Development Award and by ONR contract N00014-K-0680.

10. Bibliography

[Ahn 1986A] Ahn, I. *Towards an Implementation of Database Management Systems with Temporal Support*, in *Proceeding of the International Conference on Data Engineering*, IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 374-381.

[Ahn 1986B] Ahn, I. *Performance Modeling and Access Methods for Temporal Database Management Systems*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, Aug. 1986.

[Ahn & Snodgrass 1986] Ahn, I. and R. Snodgrass. *Performance Evaluation of a Temporal Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. C. Zaniolo. Association for Computing Machinery. Washington, DC: May 1986, pp. 96-107.

[Ahn 1987] Ahn, I. *Adative Hashing. submitted for publication*, (1987).

[Ahn & Snodgrass 1988] Ahn, I. and R. Snodgrass. *Performance Analysis of Temporal Queries. Information Sciences*, 11, June 1988.

[Bayer & McCreight 1972] Bayer, R. and E. McCreight. *Organization and Maintenance of Large Ordered Indexes. Acta Informatica*, 1, No. 3 (1972), pp. 173-189.

[Ben-Zvi 1982] Ben-Zvi, J. *The Time Relational Model*. PhD. Diss. UCLA, 1982.

[Bolour et al. 1982] Bolour, A., T.L. Anderson, L.J. Dekeyser and H.K.T. Wong. *The Role of Time in Information Processing: A Survey. SigArt Newsletter*, 80, Apr. 1982, pp. 28-48.

[Ceri & Pelagatti 1984] Ceri, S. and G. Pelagatti. *Distributed Databases Principles & Systems*. NY: McGraw-Hill, 1984.

[Christodoulakis 1987] Christodoulakis, S. *Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology. ACM Transactions on Database Systems*, 12, No. 2 (1987), pp. 137-169.

[Copeland 1982] Copeland, G. *What If Mass Storage Were Free?. IEEE Computer*, 15, No. 7, July 1982, pp. 27-35.

[Dadam et al. 1984] Dadam, P., V. Lum and H.-D. Werner. *Integration of Time Versions into a Relational Database System*, in *Proceedings of the Conference on Very Large Databases*, Ed. U. Dayal, G. Schlageter and L.H. Seng. Singapore: 1984, pp. 509-522.

[Fagin et al. 1979] Fagin, R., J. Nievergelt, N. Pippenger and H. Strong. *Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Transactions on Database Systems*, 4, No. 3, Sep. 1979, pp. 315-344.

[Fujitani 1984] Fujitani, L. *Laser Optical Disk: The Coming Revolution in On-Line Storage. Communications of the Association of Computing Machinery*, 27, No. 6, June 1984, pp. 546-554.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A Relational Data Base Management System. Proceedings of the AFIPS 1975 National Computer Conference*, 44, May 1975, pp. 409-416.

[Hoagland 1985] Hoagland, A. *Information Storage Technology: A Look at the Future. IEEE Computer*, 18, No. 7, July 1985, pp. 60-67.

[Katz & Lehman 1984] Katz, R.H. and T. Lehman. *Database Support for Versions and Alternatives of Large Design Files. IEEE Transactions on Software Engineering*, SE-10, No. 2, Mar. 1984, pp. 191-200.

[Larson 1978] Larson, P. *Dynamic Hashing. BIT*, 18 (1978), pp. 184-201.

[Litwin 1978] Litwin, W. *Virtual Hashing: A Dynamically Changing Hashing*, in *Proceedings of the Conference on Very Large Databases*, 1978, pp. 517-523.

[Litwin 1980] Litwin, W. *Linear Hashing: A New Tool For File And Table Addressing*, in *Proceedings of the Conference on Very Large Databases*, 1980, pp. 212-223.

[Lum et al. 1984] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. *Designing DBMS Support for the Temporal Dimension*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. B Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 115-130.

[Lum et al. 1985] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. *Design of an Integrated DBMS to Support Advanced Applications*, in *Proceedings of the Conference on Foundation of Data Organization*, Kyoto, Japan: May 1985.

[Maier 1985] Maier, D. *The Theory of Relational Databases*. Computer Science Press, 1985.

[McKenzie 1986] McKenzie, E. *Bibliography: Temporal Databases. ACM SIGMOD Record*, 15, No. 4, Dec. 1986, pp. 40-52.

[Morris 1968] Morris, R. *Scatter Storage Techniques. Communications of the Association of Computing Machinery*, 11, No. 1, Jan. 1968, pp. 38-43.

[Nievergelt et al. 1984] Nievergelt, J., H. Hinterberger and K. C. Sevcik. *The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems*, 9, No. 1, Mar. 1984, pp. 38-71.

[Robinson 1981] Robinson, J. *The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1981, pp. 10-18.

[Schueler 1977] Schueler, B. *Update Reconsidered*, in *Architecture and Models in Data Base Management Systems*. Ed. G. M. Nijssen. North Holland Publishing Co., 1977.

[Severance 1976] Severance, D. *Differential Files: Their Application to the Maintenance of Large Databases. ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 256-267.

[Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. *A Taxonomy of Time in Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236-246.

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. IEEE Computer*, 19, No. 9, Sep. 1986, pp. 35-42.

[Snodgrass 1987] Snodgrass, R. *The Temporal Query Language TQuel. ACM Transactions on Database Systems*, 12, No. 2, June 1987, pp. 243-298.

[Stonebraker et al. 1976] Stonebraker, M., E. Wong, P. Kreps and G. Held. *The Design and Implementation of*

INGRES. ACM Transactions on Database Systems, 1, No. 3, Sep. 1976, pp. 189-222.

[Stonebraker & Rowe 1986] Stonebraker, M. & Rowe, L. *The Design of Postgres. ACM SIGMOD Record*, , Aug. 1986, pp. 340-355.

[Yao 1977] Yao, S. *Approximating Block Accesses in Database Organizations. Communications of the Association of Computing Machinery*, 20, No. 4, Apr. 1977, pp. 260-261.


DAT
FILM